9

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

LEVEL II

(9)

TR-335 [redacted]

November 1979

Assertion Mechanisms in Programming Languages

James R. Lyle
Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, Maryland 20742

The work reported here was supported by a grant from the Air Force Office of Scientific Research [redacted]. Computer time was provided in part by the Computer Science

F49620-80-C-0001

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER **AFOSR-TR-81-0365**	2. GOVT ACCESSION NO. AD-A098069
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ASSERTION MECHANISMS IN PROGRAMMING LANGUAGES	5. TYPE OF REPORT & PERIOD COVERED INTERIM
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Marvin V. Zelkowitz ~~and~~ James R. Lyle	8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Sciences University of Maryland College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 2304/A2 61102F
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB DC 20332	12. REPORT DATE NOV [illegible]
	13. NUMBER OF PAGES 27
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Assertions typically are used to verify program behavior. However, the [illegible] of an assertion to cause a run-time exception can have practical benefits. We take this view that an exception is such an assertion failure. The implementation of assertions in a PL/I compiler is described, and the interface with the exception mechanism of PL/I (ON-units) is described. Principle usages include: test data set evaluation and extension of the domain of abstract data type specifications.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Assertion Mechanisms in Programming Languages

ABSTRACT --

Assertions typically are used to verify program behavior. However, the use of an assertion to cause a run-time exception can have practical benifits. We take this view that an exception is such an assertion failure.

The implementation of assertions in a PL/I compiler is described, and the interface with the exception mechanism of PL/I (ON-units) is described. Principle usages include: test data set evaluation and extension of the domain of abstract data type specifications.

Table Of Contents

Table of Contents

1. Introduction

Assertions, relatively new program constructs developed as part of research in program verification, typically are used to verify program behavior. They allow a programmer to make statements about what ought to be true at a point in program execution.

The language designer has several options when considering the semantics of an assertion mechanism. Originally they were considered predicates for a theorem prover to (necessarily) verify, but had no impact upon the computation process. This is compatible with the Hoare axiomatic approach towards program verification [Hoare]. Alternative views are that they indicate conditions to be tested during program execution or they could indicate a lemma (e.g. theorem, axiom, pre or post condition) to be proven by the compiler. The failure of a run-time check causes execution to stop, and may raise an exception condition.

The basic assumption in this current research is that assertions are another form of program exception, rather than simply a "bug". Many current languages include some form of exception handling (e.g. ON-units of PL/I). Or stated another way, exceptions are simply a language defined assertion (e.g. an extension to the "legality assertion" of Euclid).

In the next section of this paper, several assertion and exception mechanisms now under study are surveyed, while the

University of Maryland PLACES system assertion mechanism is explained in section 3. Section 4 discusses the goals and applications (e.g. test data evaluation and data type specification) of the PLACES mechanism and gives examples of how to make use of the facilities within PLACES.

2. Survey of Existing Languages

2.1. Exceptions

This section summarizes briefly exception handling in languages with assertion mechanisms.

2.1.1. PL/I

Exception handlers, called ON-units, are associated dynamically with exceptions. A program executes an ON statement which dynamically associates a given block with a certain exception. The block intercepts the exception as long as the block executing the ON statement is active.

At compile time it is generally impossible to identify which handler (there can be as many as the programmer wishes to write) is active at the time an exception occurs. The environment of the ON-unit is nested within that of the signaling block. Facilites in the form of builtin function calls, are available to provide more information about the exception (e.g. ONSOURCE). The ON-unit returns to the point where the exception was signaled unless the ON-unit is terminated by a GOTO.

2.1.2. Ada

The new Department of Defense language [Ada], contains an exception mechanism. Each block or program body may have an exception handler statically attached as an exit to a block. When an exception is raised the current block is terminated and control passes to the appropriate handler. If no handler is specified the exception is propagated outward to the invoking program unit (calling program or enclosing block). Unlike PL/I, control returns to the point of call after the exception has been processed.

2.1.3. Gypsy

Exceptions, called _condition clauses_ in Gypsy [Allen], are similar to the Ada design, with condition clauses statically attached to blocks. When a condition is raised, the block is terminated and the condition is processed.

2.1.4. Pascal,Euclid and ZENO

No exception handling facilities are included in these languages, although some dialects have implemented some exception capability.

2.2. Assertions

This section presents a brief description of the assertion mechanism of several typical designs. The mechanisms range from the eloborate interactive theorem prover of GYPSY to a simple run-time check (to be) generated by Ada compilers.

Most assertions are based upon axiomatic program verification [Hoare]. If P and Q are predicates and S is a statement, then

{P} S {Q}

states that if P is true (pre condition) and S is executed, then Q is true (post condition). The axiom determines what pre and post conditions are allowed.

In programs, these are usually written:

```
ASSERT P;
   S;
ASSERT Q;
```

Originally a verifier, if given the above needs to prove {P} S {Q} as a theorem. Such proofs are difficult and often impossible. Thus run-time checking was proposed as an alternative in several languages. Note that this is essentially a test of an instance of a variable, while the more formal proof is a verificatian for all possible data.

2.2.1. Pascal

The original specification of Pascal does not include assertions, however recent implementations such as [Hansen] Texas Instruments 990 minicomputer has extended Pascal to include them. Most of the recent languages (e.g. Euclid, GYPSY, Ada) are derivaties of Pascal, and do contain some assertion mechanism.

2.2.2. ADA

Ada includes an ASSERT statement, a boolean expression which must be true when the ASSERT is encountered at run-time. If the expression is false an ASSERT_ERROR exception is raised.

2.2.3. Euclid

Euclid is a systems programming language derived from Pascal with reliability and verifiability as the main design goals. Assertions are included to provide useful documentation of program specifications and to assist in program verification [Popek]. If an assertion cannot be proven at compile time a run-time check is placed in the object code by the compiler. All Euclid programs are expected to be verified, so the failure of a run-time check is a fatal error that stops the program.

The Euclid compiler is expected to pass <u>legality</u> <u>assertions</u>, conditions which must be true for a program to be legal Euclid, to the verifier whenever the compiler cannot fully check that some constraint imposed by the language is satisfied. Legality assertions are source level assertions that can be inserted by the programmer or automatically by the compiler that must be true based upon the source program. For example, for array references, legality assertions can be generated that the array index is within the array bounds.

2.2.4. ZENO

ZENO, a language strongly based on Euclid, is being designed for research in code optimization and distributed computing [Ball]. The assertion mechanism in ZENO allows specifications about what is true at a point in program execution (called point assertions) and about what is invariant over a block of code (called under assertions).

Point assertions evaluate a boolean expression. If the expression is true execution continues, otherwise the program stops. The compiler is expected to try to infer the truth of the expression from flow analysis; a run-time check is generated if the analysis cannot prove the result.

Under assertions allow invariants to be placed on blocks of code. A block labeled with an under assertion will behave as if there is a point assertion before each statement in the block and after the last statement of the block. Under assertions normally apply to all inner blocks but a relax clause can be added to an inner block to release the inner block from conforming to the under assertion.

2.2.5. GYPSY

The prime design goal of GYPSY was an integrated specification and programming language [Allen]. The assertion mechanism of GYPSY is used to express program specifications. Specifications can be placed on type parameters (REQUIRE), statements (ASSERT) and routines (ENTRY, BLOCK and EXIT). The programmer can request that the compiler prove or assume any specification. The compiler can also be directed to include a run-time check of the specification. Failure of the run-time check raises an exception.

2.2.6. PL/CS

PL/CS is a version of PL/I developed at Cornell University which "enforces some of the ideas which have come to be regarded as good programming practice" [Conway]. The PL/CS assertion mechanism includes an ASSERT boolean expression with two variations.

The FOR SOME variation attaches a DO group to the assertion. The expression must evaluate to "true" for at least one value of the index variable. For example, suppose we wish to assert that some element of the array X of dimension N is greater than zero.

We would write:

```
ASSERT (X(I) > 0) FOR SOME I = 1 TO N;
```

The FOR ALL variation is similar, however, the boolean expression must be true for all values of the index variable.

The PL/CS assertion mechanism is intended to be used as a testing aid; the assertions can be turned off by compiler option for production runs.

3. Assertions in PLACES

The PLACES project (Programming Language and Construct Evaluation System) is a research project of the University of Maryland. By using the PLUM load and go PL/I compiler [Zelkowitz a] as a basis, the compiler has been extended to integrate assertions within the exception handling capability (i.e. ON-units) of PL/I.

The model being implemented is a merger of verification and testing strategies. Similar to Euclid, the programmer sprinkles the code with assertions. A verifier tries to prove the assertions. If so, then the assertion can effectively be deleted [Figure 3-1]. If not, then the assertion remains as a run-time check. Differences from Euclid include, the failure of the run-time check to invoke an exception handler, allowing the programmer to take some action other than stopping the program.

FIGURE 3-1 PLACES Structure

```
+-----------+             +-----------+             +-----------+
|           |             |           |             |           |
|           |ASSERTIONS   |           | CANNOT      | GENERATE  |
|  PLACES   |------------>| VERIFIER  |------------>| RUN-TIME  |
|           |             |           | PROVE       |  CHECK    |
|           |             |           |             |           |
+-----------+             +-----------+             +-----------+
                               |
                               | CAN
                               | PROVE                +-----------+
                               |                      |           |
                               |        TRUE          |  DELETE   |
                               +--------------------->| ASSERTION |
                               |                      |           |
                               |                      |           |
                               |                      +-----------+
                               |
                               |
                               |                      +-----------+
                               |                      |           |
                               |        FALSE         |   PRINT   |
                               +--------------------->|   ERROR   |
                                                      |  MESSAGE  |
                                                      |           |
                                                      +-----------+
```

In many applications, programs cannot be proven. For example, given the sequence:

```
GET LIST (x);
.
.
.
```

the correctness of the program may depend upon the value of x which cannot be known until run-time. A perfect system (which we do not claim to be producing) would reduce all assertions to the following one:

```
        GET LIST (X);
        ASSERT (P(X));
```

where P(x) is the predicate that "X has the properity that ensures the correctness of the program". Thus the correctness of a program run depends on one specific run-time check. In the example of Figure 3-2 the assertions at lines 22-23 and 29 can be proven (see [Basili]) and hence deleted. What remains is P(X), where X is the vector [A,B] and P(X) is (A ≥ 0 & B ≥ 0).

3.1. Basic ASSERT statement

The basic ASSERT in PLACES has the form:

```
                ASSERT (<boolean expression>);
```

which generates a run-time test equivalent to:

```
        IF NOT (<boolean expression>) THEN SIGNAL ASSERTFAIL;
```

In other words, if the boolean expression is "true" execution continues, otherwise the ASSERTFAIL condition is raised. ASSERTFAIL was added as a new condition which can be invoked. Thus the user can write:

```
                ON ASSERTFAIL BEGIN . . . END;
```

Examples of PLACES assertions:

```
        ASSERT (X > Y);
        ASSERT (A < B & C = 3);
```

```
ASSERT (A); /* VALID ONLY IF A IS DECLARED BIT */
```

This is similar to other run-time assertion systems.

3.2. Invariant assertions

Often it is desirable to have an assertion hold over a block of code. PLACES handles this via an invariant assertion of the form:

```
ASSERT (<boolean expression>) INVARIANT;
```

Similar to ZENO, an invariant assertion generates a test of the expression before each statement at the same scope level in the remainder of the current block. The expression is also tested after the last statement in the block. It should be noted that while the invariant is checked before entry to and after exit from a block nested within the current scope, the invariant, unlike ZENO, is not checked within an inner block unless it is explicitly stated.

3.3. Named Assertions

PLACES assertions are non-executable statements in that they cannot be labeled or referenced by a GOTO statement and have no side effects. However, it is sometimes desirable to refer to a specific assertion. To this end assertions can be named. The syntax is as follows:

```
ASSERT (<name>) (<boolean expression>) (INVARIANT);
```

where the items in brackets (<name> and INVARIANT) are optional, and <name> refers to any legal PL/I identifier.

3.4. ONASSERT builtin function

Given several assertions within a program, if a user writes an ASSERTFAIL ON-unit, the ability to determine which assertion failed is needed. In order to do this, the ONASSERT builtin function was added. ONASSERT returns a character string which is the name of the assertion which raised the exception. If the failed assertion was not named the statement number of the assertion is returned.

Use of named assertions and ONASSERT leads to a model of an ON-unit that has practical applications. For example, an ASSERTFAIL ON-unit could have the structure:

```
        ON ASSERTFAIL BEGIN;
             DO CASE (ONASSERT);
                  \'LABEL1'\ DO; . . . END;
                  \'LABEL2'\ DO; . . . END;
                  .
                  .
                  .
             END; /* END CASE */
        END; /* END ON-UNIT */
```

This has a sematic structure similar to:

```
        ON ASSERTFAIL BEGIN;
             WHEN \LABEL1\ DO; . . . END;
             WHEN \LABEL2\ DO; . . . END;
             .
             .
             .
        END;
```

This structure is quite similar to GYPSY and ADA.

3.5. Execution Summary

When a PLACES program terminates a summary is printed giving for each assertion the number of executions and the number of failures.

3.6. An Example

The program in Figure 3-2, adapted from [Basili] multiplies two numbers by repeated addition. The ASSERT statements are present at lines 13, 22-23 and 29. The assertion at lines 22-23 is an example of an invariant. The expression is tested after each statement in the remainder of the enclosing block. That is, the invariant is checked after each statement between lines 23 and 2[illegible].

If any of the assertions fail the ON-unit in lines 8 to 10 is invoked. This ON-unit prints a message identifing the failed assertion and the statement where the failure occured. If the assertions at lines 13 or 29 fail ONASSERT returns the statement number of the ASSERT, which is 12 or 22 respectively. However if the ASSERT at lines 22-23 fails, ONASSERT returns LOOP_INVARIANT since the assertion is named.

Figure 3-2 PLACES Program with Assertions

```
PLUM   5:12A    <Places Project>  10/31/79  10:12:46
  1    1       MUL:PROCEDURE OPTIONS (MAIN);
  2    2          DECLARE (A,B,Y,Z);
  3    2
  4    3          DECLARE MORE_INPUT BIT (1) INIT ('1'B);
  5    3
  6    4          ON ENDFILE (SYSIN) MORE_INPUT = '0'B;
  7    4
  8    5          ON ASSERTFAIL PUT SKIP EDIT (
  9    5               'ASSERTION ',ONASSERT,' FAILED AT:',ONSTMT)
 10    5               (A,A,A,F(3));
 11    6
 12    6
 13    8          PUT SKIP LIST ('ENTER A AND B');
 14    9          GET LIST (A,B);
 15   10          DO WHILE (MORE_INPUT);
 16   11               PUT SKIP DATA (A,B);
 17   11               /*COMPUTE Z := A*B BY ADDITION */
 18   12               ASSERT (A >= 0 & B >= 0);
 19   13               Z = 0;
 20   14               Y = B;
 21   15               BEGIN;
 22   16                    ASSERT LOOP_INVARIANT
 23   16                        (Z = A*(B-Y) & Y >= 0) INVARIANT;
 24   17                    DO WHILE (Y>0);
 25   18                              Y = Y - 1;
 26   19                              Z = Z + A;
 27   20                    END;
 28   21               END;
 29   22               ASSERT (Z = A*B);
 30   23               PUT SKIP EDIT (A,' * ',B,' = ',Z)
 31   23                    (F(3),A,F(3),A,F(4));
 32   23
 33   24               PUT SKIP LIST ('ENTER A AND B');
 34   25               GET LIST (A,B);
 35   26          END;
 36   27       END MUL;
WARNING CG  52 ASSERTFAIL is non-standard PL/1
WARNING CG  52 ONASSERT is non-standard PL/1
WARNING CG  52 ONSTMT is non-standard PL/1
WARNING CG  52 ASSERT is non-standard PL/1
COMPILE TIME      878 MSEC.
```

Figure 3-3 Output from Program MUL

```
ENTER A AND B
A=              2           B=             3;
ASSERTION LOOP_INVARIANT FAILED AT: 19
ASSERTION LOOP_INVARIANT FAILED AT: 19
ASSERTION LOOP_INVARIANT FAILED AT: 19
  2 *   3 =    6
ENTER A AND B
A=              8           B=             5;
ASSERTION LOOP_INVARIANT FAILED AT: 19
ASSERTION LOOP_INVARIANT FAILED AT: 19
ASSERTION LOOP_INVARIANT FAILED AT: 19
ASSERTION LOOP_INVARIANT FAILED AT: 19
ASSERTION LOOP_INVARIANT FAILED AT: 19
  8 *   5 =   40
 EX  23 Normal exit
ASSERTION SUMMARY:
          STMT #     EXECUTIONS        FAILURES
               12            2
               15 INV       38                  8
               22            2
EXECUTION TIME       1318 MSEC.
```

3.7. Enhancements Under study

Additional features to PLACES are still under study; however, since they can be simulated relatively easily within the current structure, the implementation has been postponed.

After evaluation these features may be added to the language:

1) Initial values. $X corresponds to the initial value of X on entry to a block.

X can only be tested within an ASSERT statement. Now the programmer can simoly declare a variable and initilize it in order to simulate this.

2) FOR ALL and FOR SOME. These constraints (like in PL/CS) are also being considered. They can be simulated as function calls within expressions in ASSERT statements.

4. Using Assertions in Programs

4.1. Provide Information to a Compiler

Assertions can be used to write specifications which a compiler tries to prove. This has not been very fruitful in general because of the difficulty of writing good theorem provers. However, many simple assertions and special cases can be checked at compile time with techniques such as data flow analysis.

4.2. Program Testing

Assertions can be used in program testing to verify pre and post conditions and to monitor constant relationships i.e., invariants. Some general rules of thumb should be followed when using assertions in program testing (some of which can be enforced by the compiler).

1) Assertions should not be referenced except by some other component of the assertion mechanism such as an ASSERTFAIL ON-unit.

2) Assertions should be used so that they can be deactivated without changing the meaning or results of a program. They should not be used to screen input for validity, for example.

Of course, these guidelines do not apply when assertions are use for some purpose other than testing.

4.2.1. Pre and Post Conditions

One method of using Assertions is as pre conditions or post conditions on logical groups of statements. When used as a pre condition an assertion verifies that the previous statements have executed correctly, i.e., it is a post condition on the statements already executed.

4.2.2. Invariant Assertions

Another usage of assertions is to monitor a relationship among variables which must remain constant over a block of statements. This can be done by an invariant assertion as in lines 22-23 of Figure 3-2. It should be pointed out that invariant assertions cannot in general be used to specify loop invariants in the Hoare sense since a loop invariant often fails to hold within the body of a loop.

4.2.3. Test Data Evaluation

Assertions can be used to help evaluate the thoroughness of a test data set. By placing assertions along each control path the assertion summary of PLACES identifies which paths have not been executed.

For example, in Figure 4-1 the execution count of the assertion WHILE1 is the number of times the while loop is reached during execution. The failure count is the number of times the loop is skipped. So by examining the execution summary the programmer knows if the test data set has exercised the loop or skipped it or both.

Figure 4-1 Using Assertions in Path Testing

```
        ASSERT WHILE1 (A < B);
    LOOP:DO WHILE (A < B);
        .
        .
        .
        END;
```

Of course, even if all paths of a program are executed by a test set there can still be errors in the code [Goodenough]. The main thrust of program testing is to expose program errors. It is believed that the more components of a program that are exercised the lower the chance of undetected errors. A careful selection of assertions can provide more information than path testing alone.

In Figure 4-2 the assertions OR1, OR2 and OR3 indicate the combination of conditions leading to execution of the THEN path of IF1. IF1 is an example of monitoring a disjunction of two expressions. An example of monitoring a conjunction of two expressions is presented in IF2.

Figure 4-2 Monitoring Path Selection

```
   DECLARE GOT_HERE BIT INIT ('1'B);
   IF1:IF (A > B OR C < D) THEN DO;
            ASSERT OR1 (A > B & C < D);
            ASSERT OR2 (A > B);
            ASSERT OR3 (C < D);
            .
            .
            .
            END;
       ELSE DO;
            ASSERT (GOT_HERE);
            .
            .
            .
            END;
  IF2:IF (A > B & C < D) THEN DO;
            ASSERT (GOT_HERE);
            .
            .
            .
            END;
       ELSE DO;
            ASSERT (NOT (A > B OR C < D));
            ASSERT (NOT (A > B));
            ASSERT (NOT (C < D));
            .
            .
            .
            END;
```

4.3. Exceptions Viewed as Assertions

We have been viewing assertion failures as one out of several classes of exceptional conditions; usually they are signifing an incorect program. However, this is in fact backwards, other exceptions are in reality assertion failures. Any language has certain data type constraints (e.g. division is with a non-zero divisor), and violating those constraints raises an exception. We believe that this is a more reasonable viewpoint. Consider, for example, the statement I = J + K. When the programmer writes such a statement she (or at least the compiler) has in mind the assertion:

MIN_INTEGER ≤ (J + K) AND (J + K) ≤ MAX_INTEGER

where MIN_INTEGER and MAX_INTEGER are the smallest and largest integers representable within the implementation. If this implied assertion fails an overflow exception is raised. This sort of implied assertion has been extended as the legality assertion of Euclid.

With this view we have more flexibility in thinking about, designing and using an assertion mechanism. We can have assertions check for boundary conditions and process these special cases elsewhere in an exception handler. Our code is not cluttered by the details of handling special cases, as the following section demonstrates.

4.4. Data Type Specifications

In an earlier paper [Zelkowitz b], an extension to PLUM has been described which implements abstract data types called ENVIRONMENT variables, a form of pointer variable implementation with protection against invalid usages. The model that was implemented has the following structure. For a data type STACK with operations PUSH and POP, the source code implementing the abstraction would be:

```
   STACK:ABSTRACTION;
       DECLARE 1 STACK,
       2  STORAGE (100),          /* STACK IS AN ARRAY */
       2  CURRENT_PTR INIT (0),
       2  SIZE INIT (100);

       PUSH:FUNCTION (X,Y);       /* PUSH Y ONTO X */
           DECLARE X ENV(STACK);
           DECLARE Y;
           . . .
       END PUSH;
       POP:FUNCTION (X);          /* POP X AND RETURN TOP VALUE */
       . . .
       END POP;
   END STACK;
```

A user of a stack would code:

```
       DECLARE X ENV (STACK);
       . . .
       CALL PUSH (X,I);
       . . .
       I = POP(X);
```

and would only have access to the data type name STACK and the operations PUSH and POP.

Assertions fill the role of specifications for this model. For example, the PUSH operation could have the syntax:

```
   PUSH:FUNCTION (X,Y);
       ASSERT PUSH_FAIL (X.CURRENT_PTR < X.SIZE);
       . . .
   END PUSH;
```

Thus the implementation need not be concerned with improper calls on PUSH since the assertion ensures (either through a proof or a run-time check) that the condition cannot arise. The full implementation of a STACK can then be coded as:

```
   STACK:ABSTRACTION;
       ON_UNIT:PROCEDURE;
           WHEN PUSH_FAIL BEGIN . . . END;
           WHEN POP_FAIL  BEGIN . . . END;
           . . .
```

```
        END ON_UNIT;
        . . .
        PUSH:FUNCTION (X,Y);
        ON ASSERTFAIL CALL ON_UNIT;
        ASSERT PUSH_FAIL <expression>;
        . . .
        END PUSH;
    END STACK;
```

One implementation detail under consideration is to automatically execute the statement:

```
ON ASSERTFAIL CALL ON_UNIT;
```

on entry to the function, thus have exceptions handled automatically. This leads to a practical system that has many of the same characteristics as ALPHARD [Wulf] in a practical system. The ON statement in procedure PUSH can automatically be generated if there is an exception block at the head of the procedure. These ideas are now under development.

5. References

[ADA] Cii Honeywell Bull, "Preliminary Ada Reference Manual", SIGPLAN NOTICES, vol. 14, no. 6, June 1979.

[Allen] Allen L. Ambler, Donald I. Good and Wilhelm F. Burger, "Report on the Language Gypsy", ICSCA-CMP-1, Institute for Computing Science and Computer Applications, The University of Texas at Austin, August 1976.

[Ball] J. E. Ball, J. R. Low and G. J. Williams, "Preliminary ZENO Language Description", SIGPLAN NOTICES, vol. 14, no. 9, p. 17-34, September 1979.

[Basili] Victor R. Basili and Robert E. Noonan, "A Comparison of the Axiomatic and Functional Models, of Structured Programming", University of Maryland Computer Science Technical Report TR-630, February 1978.

[Conway] Richard Conway, A Primer on Disciplined Programming, Winthrop Publishers, 1978.

[Goodenough] John B. Goodenough and Susan L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, vol. SE-1, no. 2, p. 156-173, June 1975.

[Hansen] G. J. Hansen, G. A. Shoults and J. D. Coinmeat, "Construction of a Transportable, Multi-Pass Compiler, for Extended Pascal", Proceedings of the SIGPLAN Symposium on Compiler Construction, SIGPLAN NOTICES vol. 14, no. 8, p. 117-126, August 1979.

[Hoare] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming", CACM, vol. 12, no. 10, p. 576-583, October 1969.

[Popek] G. J. Popek, J. J. Horning, B. W. Lampson, J. G. Mitchell and R. L. London, "Notes on the Design of Euclid", Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN NOTICES vol. 12, no. 3, p. 11-18, March 1977.

[Wulf] W. A. Wulf, R. L. London and M. Shaw, "An Introduction to the Construction and Verification of ALPHARD Programs", IEEE Transactions on Software Engineering, vol. 2, no. 4, p. 253-265, 1976.

[Zelkowitz a] Marvin V. Zelkowitz, "PLACES: Programming Language and Construct Evaluation System", Seventeenth Annual Technical

Symposium, National Bureau of Standards, Gaithersburg Md. June 1975.

[Zelkowitz b] Marvin V. Zelkowitz and Howard J. Larsen, "Implementation of a Capability Based Data Abstraction", IEEE Transactions on Software Engineering, vol. 4, no. 1, p. 56-64, January 1978.

END

DATE

ILME